VIT

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Information Technology & Engineering
Department of Computer Application

Winter Semester 2023

Soft Computing

Digital Assignment-1

Submitted By:
Srijan Paria
22MCA0266
Shreya Saini
22MCA0275
Rushil Awasthi
22MCA0276

22MCA0285
SUBRAMANIAN S
22MCA0289

Ques. Apply Stacked LSTM, Multiplicative LSTM, Bidirectional GRU

DATASET:

https://www.kaggle.com/datasets/meetnagadia/walmart-stock-price-from-19722022

OBJECTIVE:

To explore and leverage the capabilities of above advanced recurrent neural network architectures for time series forecasting or sequential data analysis tasks.

Importing several libraries and defining some classes and functions related to time series forecasting using deep learning models

```
[1] import pandas as pd
    import numpy as np
    import math
    from sklearn.metrics import mean_squared_error
    from tensorflow.keras.models import load_model
    from sklearn.preprocessing import MinMaxScaler
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import LSTM, Dense, Bidirectional, GRU
```

Load the dataset

```
[3] # Load the dataset
    data = pd.read_csv('/content/drive/MyDrive/DA/WMT.csv')
```

Data Preprocessing

Drop column 'Adj Close' from the dataset, Because specified column is not used in predicting the close value of shares on a particular day.

```
[ ] data = data.drop('Adj Close', axis=1)
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(data['Close'].values.reshape(-1, 1))
```

Printing first five rows of table

```
[ ] print(data.head())
```

```
         Date      Open      High       Low     Close   Volume
0  1972-08-25  0.063477  0.064697  0.063477  0.064453  2508800
1  1972-08-28  0.064453  0.064941  0.064209  0.064209   972800
2  1972-08-29  0.063965  0.063965  0.063477  0.063477  1945600
3  1972-08-30  0.063477  0.063477  0.062988  0.063477   409600
4  1972-08-31  0.062988  0.062988  0.062500  0.062500   870400
```

Splitting the dataset into 70 percent training and 30 percent testing data

```
train_size = int(len(scaled_data) * 0.7)
train_data = scaled_data[:train_size]
test_data = scaled_data[train_size:]
```

Defining a function create_dataset and using it to create training and testing datasets for time series forecasting.

```
[ ] def create_dataset(dataset, time_steps=1):
        X, Y = [], []
        for i in range(len(dataset) - time_steps):
            X.append(dataset[i:i + time_steps, 0])
            Y.append(dataset[i + time_steps, 0])
        return np.array(X), np.array(Y)

    time_steps = 60  # Adjust this value according to your needs

    X_train, Y_train = create_dataset(train_data, time_steps)
    X_test, Y_test = create_dataset(test_data, time_steps)
```

Reshape the input data to be in the form [samples, time steps, features]

```
[ ] X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
    X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
```

1. Stacked LSTM

```
[ ] # Stacked LSTM
    stacked_lstm_model = Sequential()
    stacked_lstm_model.add(LSTM(units=50, return_sequences=True, input_shape=(time_steps, 1)))
    stacked_lstm_model.add(LSTM(units=50, return_sequences=True))
    stacked_lstm_model.add(LSTM(units=50))
    stacked_lstm_model.add(Dense(units=1))
    stacked_lstm_model.compile(optimizer='adam', loss='mean_squared_error')
    stacked_lstm_model.fit(X_train, Y_train, epochs=5, batch_size=32)
```

```
Epoch 1/5
272/272 [==============================] - 39s 117ms/step - loss: 6.4541e-04
Epoch 2/5
272/272 [==============================] - 33s 122ms/step - loss: 8.1201e-05
Epoch 3/5
272/272 [==============================] - 32s 116ms/step - loss: 7.1117e-05
Epoch 4/5
272/272 [==============================] - 32s 118ms/step - loss: 5.7599e-05
Epoch 5/5
272/272 [==============================] - 33s 120ms/step - loss: 5.7047e-05
<keras.callbacks.History at 0x7f65cf095ff0>
```

```
[ ] # Save the Stacked LSTM model
    stacked_lstm_model.save('stacked_lstm_model.h5')
```

calculating the root mean squared error (RMSE) between the predicted values and the actual values of the target variable.

```
[ ] stacked_lstm_model = load_model('stacked_lstm_model.h5')
    # Calculate RMSE for Stacked LSTM model
    stacked_lstm_predictions = stacked_lstm_model.predict(X_test)
    stacked_lstm_rmse = math.sqrt(mean_squared_error(Y_test, stacked_lstm_predictions))
    stacked_lstm_accuracy = 100 - stacked_lstm_rmse  # Example accuracy metric

    print("Stacked LSTM Accuracy:", stacked_lstm_accuracy)
```

```
116/116 [==============================] - 6s 36ms/step
Stacked LSTM Accuracy: 99.97531614720381
```

2. Multiplicative LSTM

```
[ ] # Multiplicative LSTM
    multiplicative_lstm_model = Sequential()
    multiplicative_lstm_model.add(LSTM(units=50, return_sequences=True, input_shape=(time_steps, 1)))
    multiplicative_lstm_model.add(LSTM(units=50, return_sequences=True, recurrent_activation='sigmoid'))
    multiplicative_lstm_model.add(LSTM(units=50, recurrent_activation='sigmoid'))
    multiplicative_lstm_model.add(Dense(units=1))
    multiplicative_lstm_model.compile(optimizer='adam', loss='mean_squared_error')
    multiplicative_lstm_model.fit(X_train, Y_train, epochs=5, batch_size=32)

    # Save the Multiplicative LSTM model
    multiplicative_lstm_model.save('multiplicative_lstm_model.h5')
```

```
Epoch 1/5
272/272 [==============================] - 33s 98ms/step - loss: 5.4979e-04
Epoch 2/5
272/272 [==============================] - 26s 97ms/step - loss: 7.1800e-05
Epoch 3/5
272/272 [==============================] - 27s 99ms/step - loss: 8.2068e-05
Epoch 4/5
272/272 [==============================] - 27s 100ms/step - loss: 7.0768e-05
Epoch 5/5
272/272 [==============================] - 27s 99ms/step - loss: 6.5035e-05
```

Calculate RMSE for Multiplicative LSTM model

```
[ ] multiplicative_lstm_predictions = multiplicative_lstm_model.predict(X_test)
    multiplicative_lstm_rmse = math.sqrt(mean_squared_error(Y_test, multiplicative_lstm_predictions))
    multiplicative_lstm_accuracy = 100 - multiplicative_lstm_rmse  # Example accuracy metric

    print("Multiplicative LSTM Accuracy:", multiplicative_lstm_accuracy)
```

```
116/116 [==============================] - 5s 27ms/step
Multiplicative LSTM Accuracy: 99.96804810076364
```

3. Bidirectional GRU

```
[ ] # Bidirectional GRU
    bidirectional_gru_model = Sequential()
    bidirectional_gru_model.add(Bidirectional(GRU(units=50, return_sequences=True), input_shape=(time_steps, 1)))
    bidirectional_gru_model.add(Bidirectional(GRU(units=50)))
    bidirectional_gru_model.add(Dense(units=1))
    bidirectional_gru_model.compile(optimizer='adam', loss='mean_squared_error')
    bidirectional_gru_model.fit(X_train, Y_train, epochs=5, batch_size=32)

    # Save the Bidirectional GRU model
    bidirectional_gru_model.save('bidirectional_gru_model.h5')
```

```
Epoch 1/5
272/272 [==============================] - 47s 139ms/step - loss: 3.7456e-04
Epoch 2/5
272/272 [==============================] - 37s 135ms/step - loss: 3.3352e-05
Epoch 3/5
272/272 [==============================] - 38s 139ms/step - loss: 2.7667e-05
Epoch 4/5
272/272 [==============================] - 36s 134ms/step - loss: 2.6192e-05
Epoch 5/5
272/272 [==============================] - 39s 142ms/step - loss: 2.3048e-05
```

Calculating RMSE for Bidirectional GRU model

```
[ ] # Calculate RMSE for Bidirectional GRU model
    bidirectional_gru_predictions = bidirectional_gru_model.predict(X_test)
    bidirectional_gru_rmse = math.sqrt(mean_squared_error(Y_test, bidirectional_gru_predictions))
    bidirectional_gru_accuracy = 100 - bidirectional_gru_rmse  # Example accuracy metric

    print("Bidirectional GRU Accuracy:", bidirectional_gru_accuracy)
```

```
116/116 [==============================] - 5s 27ms/step
Bidirectional GRU Accuracy: 99.98699211988969
```

Evaluating the loss of different models (Stacked LSTM, Multiplicative LSTM, and Bidirectional GRU) on the testing dataset.

```
[ ] # Evaluate the models
    stacked_lstm_loss = stacked_lstm_model.evaluate(X_test, Y_test)
    multiplicative_lstm_loss = multiplicative_lstm_model.evaluate(X_test, Y_test)
    bidirectional_gru_loss = bidirectional_gru_model.evaluate(X_test, Y_test)

    print("Stacked LSTM Loss:", stacked_lstm_loss)
    print("Multiplicative LSTM Loss:", multiplicative_lstm_loss)
    print("Bidirectional GRU Loss:", bidirectional_gru_loss)

    116/116 [==============================] - 7s 35ms/step - loss: 6.0929e-04
    116/116 [==============================] - 5s 32ms/step - loss: 0.0010
    116/116 [==============================] - 4s 23ms/step - loss: 1.6920e-04
    Stacked LSTM Loss: 0.0006092925905250013
    Multiplicative LSTM Loss: 0.0010209238389506936
    Bidirectional GRU Loss: 0.00016920491179917008
```

CONCLUSION:

By comparing the loss values in the above code snippet, we observe that bidirectional GRU has least loss value which is 0.00017. Moreover, the computational time of bidirectional GRU is also less than stacked LSTM and multiplicative LSTM, hence we conclude that bidirectional GRU had the best performance on the testing dataset.